Dense Arithmetic over Finite Fields with CUMODP

Sardar Anisul Haque[1] Xin Li[2] Farnam Mansouri[1]
Marc Moreno Maza[1] Wei Pan[3] Ning Xie[1]

[1]University of Western Ontario, Canada
[2]Universidad Carlos III, Spain
[3]Intel Corporation, Canada

ICMS, August 8, 2014

Outline

Background

Reducing everything to multiplication

- Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation.
- Algebraic complexity is often estimated in terms of multiplication time
- At the software level, this reduction is also common (Magma, NTL)
- Can we do the same for SIMD-multithreaded algorithms?

Building blocks in scientific software

- The *Basic Linear Algebra Subprograms* (BLAS) is an inspiring and successful project providing low-level kernels in linear algebra, used by LINPACK, LAPACK, MATLAB, Mathematica, Julia (among others).
- Other BB's successful projects: FFTW, SPIRAL (among others).
- The *The GNU Multiple Precision Arithmetic Library* project plays a similar role for rational numbers and floating-point numbers.
- No symbolic computation software dedicated to *sequential* polynomial arithmetic managed to play the unification role of the BLAS.
- Could this work in the case of GPUs?

CUMODP Mandate (1/2)

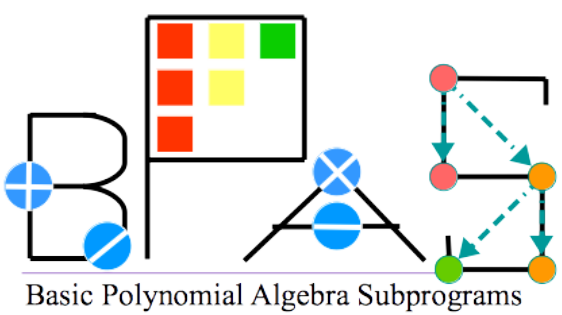
Functionalities

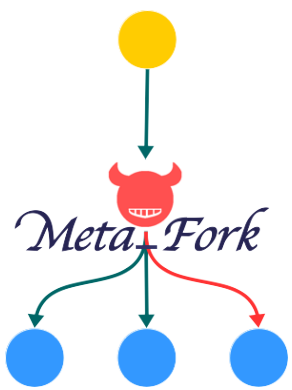
- Level 1: basic arithmetic operations that are specific to a polynomial representation or a coefficient ring: multi-dimensional FFTs/TFTs, converting integers from CRA to mixed-radix representations
- Level 2: basic arithmetic operations for dense or sparse polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{Z}$ or in floating point numbers: polynomial multiplication, polynomial division
- Level 3: advanced arithmetic operations on families of polynomials: operations based on subproduct trees, computations of subresultant chains.

CUMODP Mandate (2/2)

Targeted architectures

- Graphics Processing Units (GPUs) with code written in `CUDA`,
- CUMODP aims at supporting BPAS (Basic Polynomial Algebra Subprograms) which targets multi-core processors and which is written in `CilkPlus`.
- Thanks to our `Meta_Fork` framework, automatic translation between `CilkPlus` and `OpenMP` are possible, as well as conversions to C/C++.
- Unifying code for CUMODP and BPAS is conceivable (see the `SPIRAL` project) but highly complex (multi-core processors enforce memory consistency while GPUs do not, etc.)





CUMODP Design

Algorithm choice

- Level 1 functions (n-D FFTs/TFTs) are highly optimized in terms of arithmetic count, locality and parallelism.
- Level 2 functions provide several algorithms or implementation for the same operation: coarse-grained & fine-grained, plain & FFT-based.
- Level 3 functions combine several Level 2 algorithms for achieving a given task.

Implementation techniques

- At Level 2, the user can choose between algorithms minimizing work or algorithms maximizing parallelism
- At Level 3, this leads to adaptive algorithms that select appropriate Level 2 functions depending on available resources (number of cores, input data size).

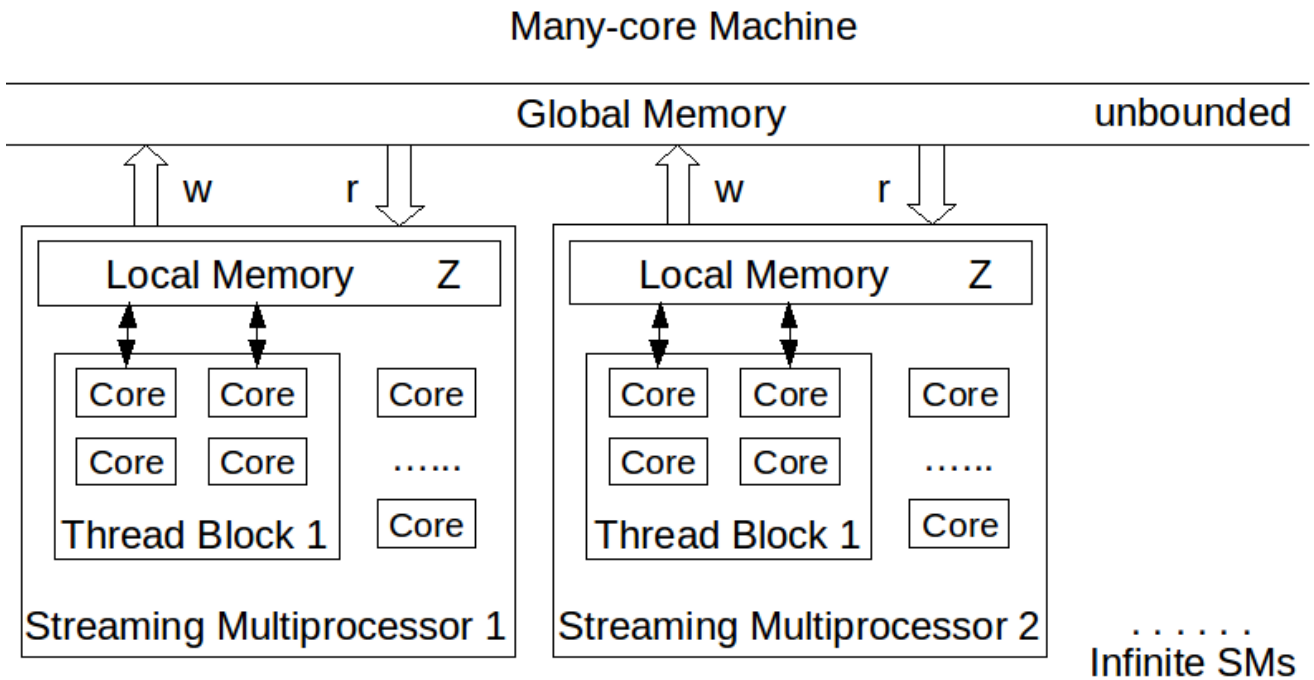
Code organization

Main features

- developers and users have access to the same code.
- mainly 32bit arithmetic so far
- Regression tests and benchmark scripts are also distributed.
- Documentation is generated by `doxygen`.
- A manually written documentation is work in progress.
- Three student theses and about 10 papers present & document CUMODP.

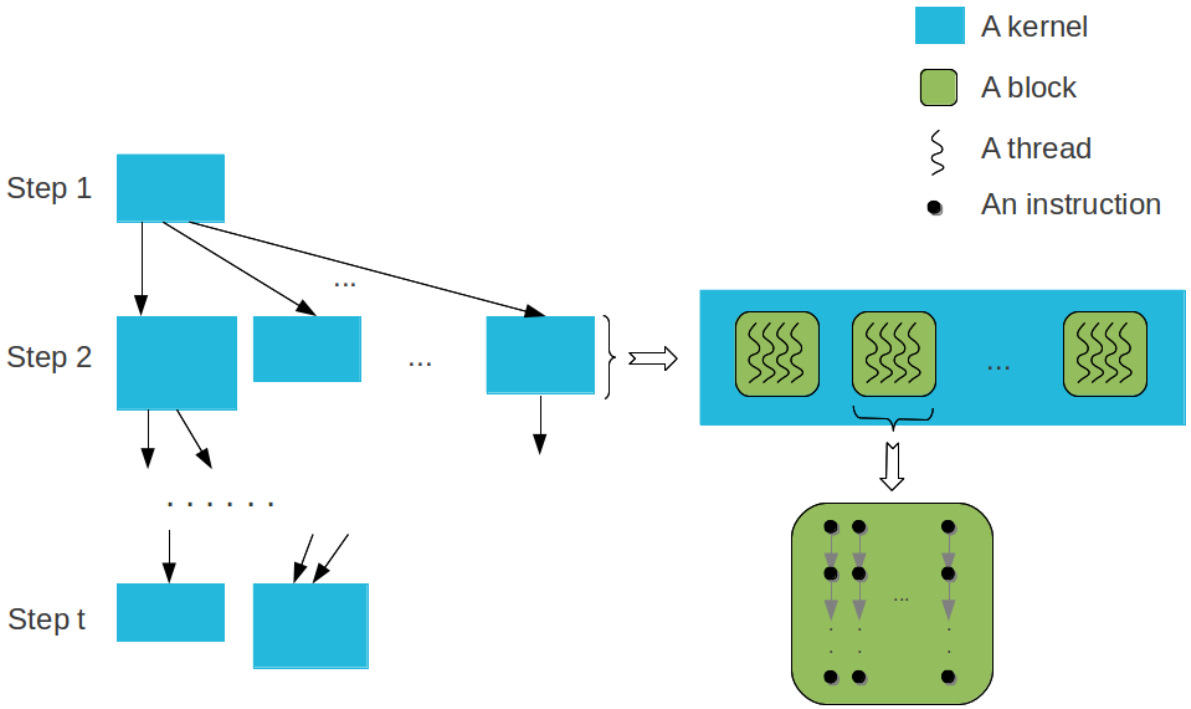
Outline

A many-core machine model



Similarly to a CUDA program, an MMM program specifies for each kernel the number of thread-blocks and the number of threads per thread-block.

A many-core machine model: programs



An MMM program $\mathcal{P}$ is a directed acyclic graph (DAG), called **kernel DAG** of $\mathcal{P}$, whose vertices are kernels and edges indicate serial dependencies.

Since each kernel of the program $\mathcal{P}$ decomposes into a finite number of thread-blocks, we map $\mathcal{P}$ to a second graph, called **thread-block DAG** of $\mathcal{P}$, whose vertex set consists of all thread-blocks of $\mathcal{P}$.

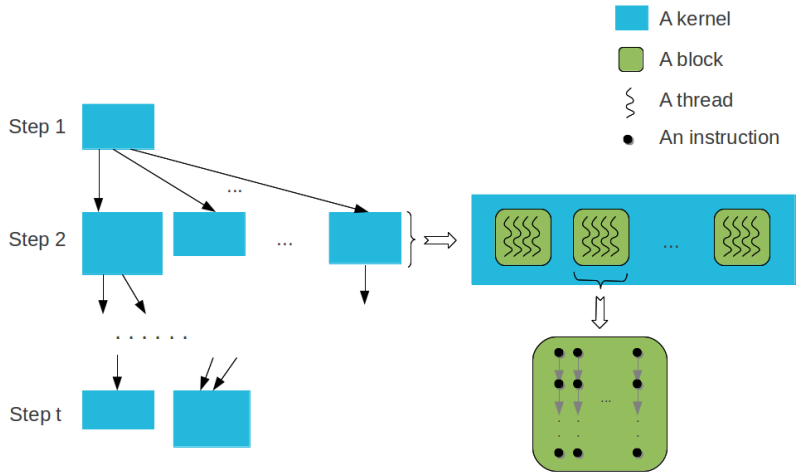
A many-core machine model: machine parameters

Machine parameters

- Z: Size (expressed in machine words) of the local memory of any SM.
- U: Time (expressed in clock cycles) to transfer one machine word between the global memory and the local memory of any SM. (Assume $U > 1$.)

Thus, if α and β are the maximum number of words respectively read and written to the global memory by a thread of a thread-block, then the data transferring time T_D between the global and local memory of a thread-block satisfies $T_D \leq (\alpha + \beta)\, U$.

A many-core machine model: complexity measures (1/2)



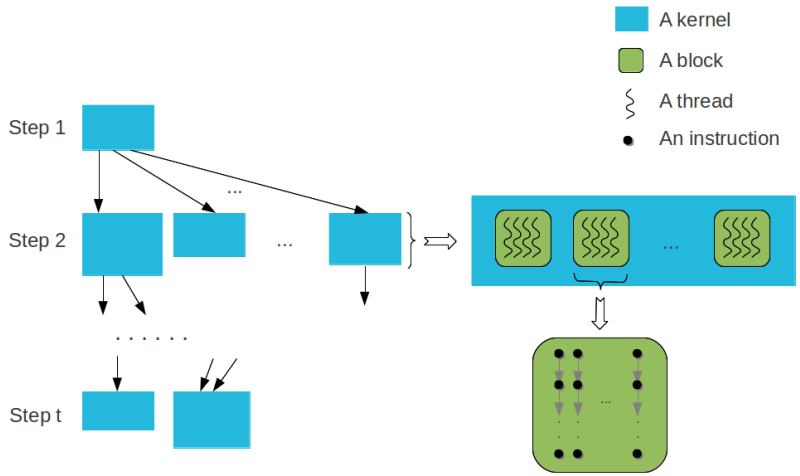
For any kernel $\mathcal{K}$ of an MMM program,

- **work** $W(\mathcal{K})$ is the total number of local operations of all its threads;
- **span** $S(\mathcal{K})$ is the maximum number of local operations of one thread.

For the entire program $\mathcal{P}$,

- **work** $W(\mathcal{P})$ is the total work of all its kernels;
- **span** $S(\mathcal{P})$ is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG.

A many-core machine model: complexity measures (2/2)



For any kernel $\mathcal{K}$ of an MMM program,

- **parallelism overhead** $O(\mathcal{K})$ is the total data transferring time among all its thread-blocks.

For the entire program $\mathcal{P}$,

- **parallelism overhead** $O(\mathcal{P})$ is the total parallelism overhead of all its kernels.

A many-core machine model: running time estimates

A Graham-Brent theorem with parallelism overhead

Theorem. Let K be the maximum number of thread blocks along an anti-chain of the thread-block DAG of $\mathcal{P}$. Then the running time $T_{\mathcal{P}}$ of the program $\mathcal{P}$ satisfies:

$$T_{\mathcal{P}} \leq (N(\mathcal{P})/\mathrm{K} + L(\mathcal{P}))\, C(\mathcal{P}),$$

where

$N(\mathcal{P})$: number of vertices in the thread-block DAG,

$L(\mathcal{P})$: critical path length (where length of a path is the number of edges in that path) in the thread-block DAG.

Note: The Graham-Brent theorem: , a greedy scheduler operating with P processors executes a multithreaded computation with work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty$$

.

Outline

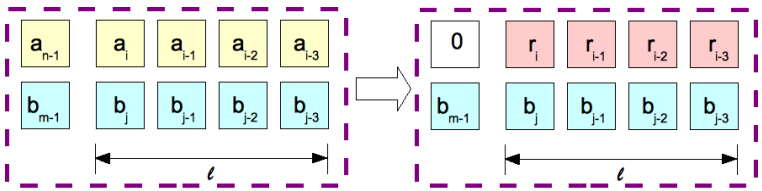
Applying the MCM model

Applying MCM to minimize parallelism overheads by determining an appropriate value range for a given program parameter

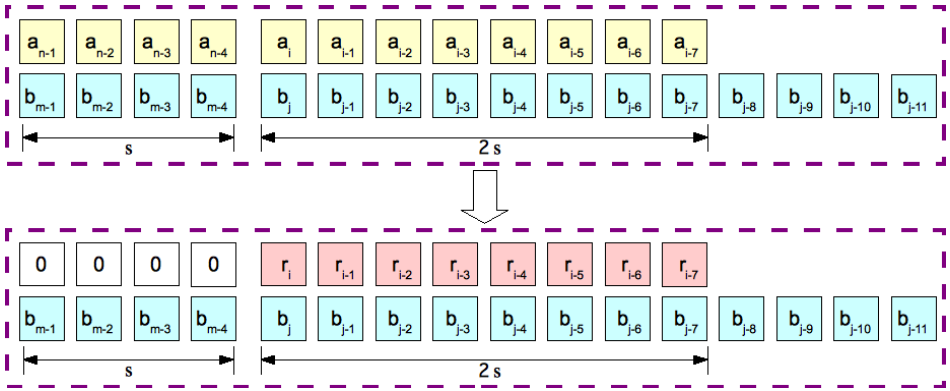
- In each case, a program $\mathcal{P}(s)$ depends on a parameter s which varies in a range $\mathcal{S}$ around an initial value s_0;
- the work ratio W_{s_0}/W_s remains essentially constant, meanwhile the parallelism overhead O_s varies more substantially, say $O_{s_0}/O_s \in \Theta(s - s_0)$;
- Then, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the ratio O_{s_0}/O_s.
- Next, we use our version of Graham-Brent theorem to check whether the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_o)$. If this holds, we view $\mathcal{P}(s_{\min})$ as a solution of our problem of algorithm optimization (in terms of parallelism overheads).

MCM applied to plain univariate polynomial division

Applying MCM to the polynomial division operation



Naive division algorithm of a thread-block with $s = 1$: each kernel performs 1 division step



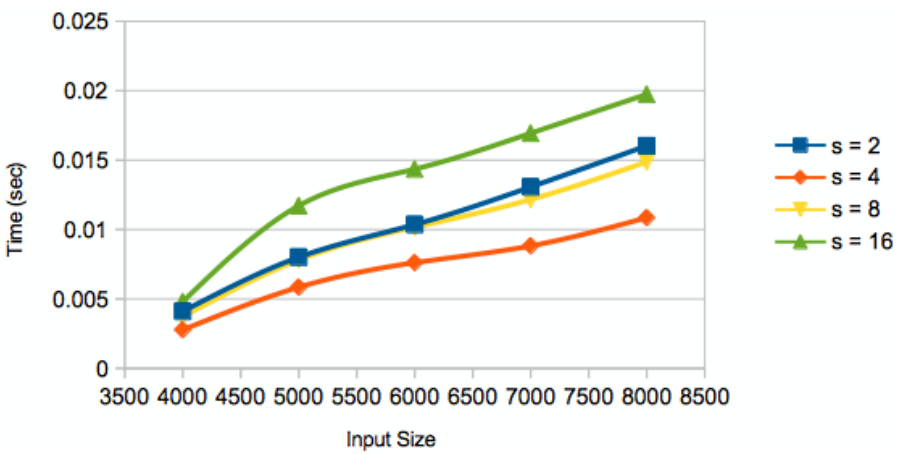
Optimized division algorithm a thread-block with $s > 1$: each kernel performs s division steps

- $\frac{W_1}{W_s} = \frac{8\,(Z+1)}{9\,Z+7}, \quad \frac{O_1}{O_s} = \frac{20}{441}Z, \quad \frac{T_1}{T_s} = \frac{(3+5\,U)\,Z}{3\,(Z+21\,U)}.$
- $T_1/T_s > 1$ if and only if $Z > 12.6$ holds, which clearly holds on actual GPU architectures.
- Thus, the optimized algorithm (that is for $s > 1$) is overall better than the naive one (that is for $s = 1$).

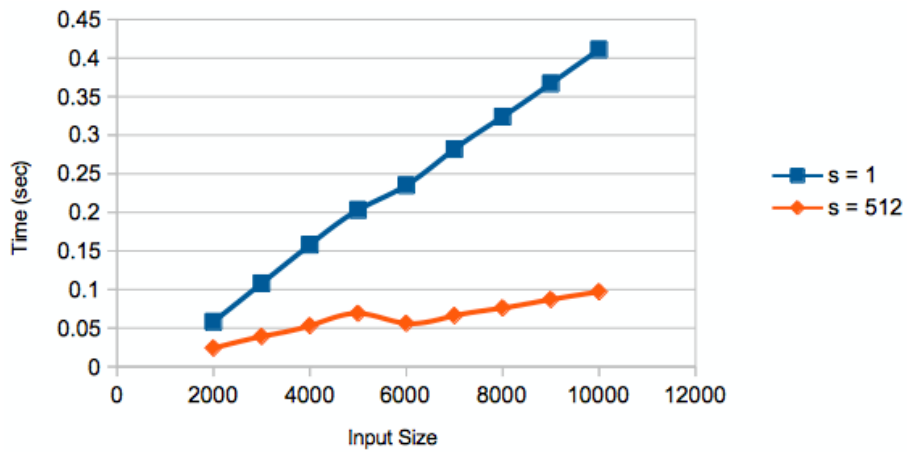
MCM applied to plain univariate polynomial multiplication and the Euclidean algorithm (1/2)

Applying MCM to the plain multiplication and the Euclidean algorithm

- For plain polynomial multiplication, this analysis suggested to minimize s.
- For the Euclidean algorithm, our analysis suggested to maximize the program parameter s.
- Both are verified experimentally.



Plain polynomial multiplication

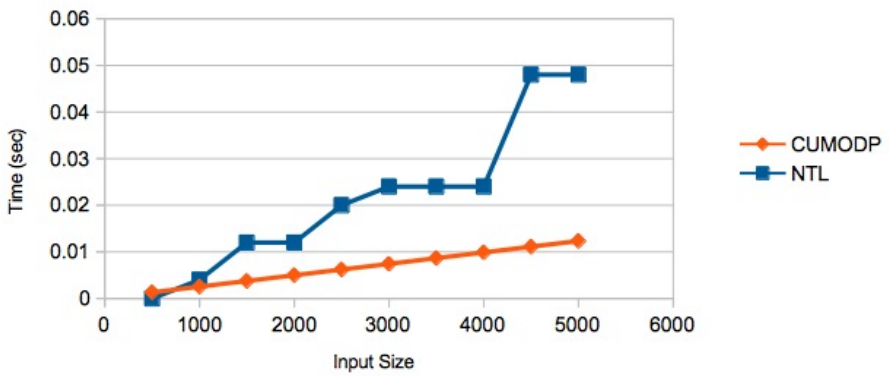


The Euclidean algorithm

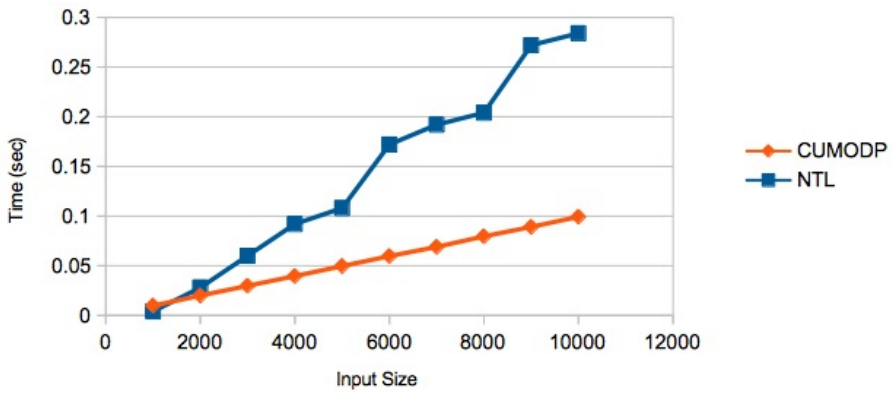
MCM applied to plain univariate polynomial multiplication and the Euclidean algorithm (1/2)

CUMODP vs NTL

- CUMODP: plain but parallel algorithms
- NTL: asymptotically fast FFT-based but serial algorithms
- Our experimentations are executed on a NVIDIA Tesla M2050 GPU and an Intel Xeon X5650 CPU.



CUMODP plain polynomial division vs NTL FFT-based polynomial division.



CUMODP plain Euclidean algorithm vs NTL FFT-based polynomial GCD.

FFT-based multiplication

Our GPU implementation relies on Stockham FFT algorithm

- Let d be the degree, then $n = 2^{\lceil \log_2(2\,d-1) \rceil}$.
- Based on the MCM, the work is $15\,n\,\log_2(n) + 2\,n$, the span is $15\,n + 2$, and the parallelism overhead is $(36\,n + 21)\,U$.

Size	CUMODP	FLINT	Speedup
2^{12}	0.0032	0.003	
2^{13}	0.0023	0.008	3.441
2^{14}	0.0039	0.013	3.346
2^{15}	0.0032	0.023	7.216
2^{16}	0.0065	0.045	6.942
2^{17}	0.0084	0.088	10.475
2^{18}	0.0122	0.227	18.468
2^{19}	0.0198	0.471	23.738
2^{20}	0.0266	1.011	27.581
2^{21}	0.0718	2.086	29.037
2^{22}	0.1451	4.419	30.454
2^{23}	0.3043	9.043	29.717

Table: Running time (in sec.) for FFT-based polynomial multiplication: CUMODP vs

Outline

Parallelizing subproduct-tree technniques

Challenging in parallel implementation:

- The divide-and-conquer formulation of operations is not sufficient to provide enough parallelism.
- One must parallelize the underlying polynomial arithmetic operations.
- The degrees of the involved polynomials vary greatly during the course of the execution of operations (subproduct tree, evaluation or interpolation).
- So does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.

Subproduct trees

Subproduct tree technique

- Split the point set $U = \{u_0, \ldots, u_{n-1}\}$ with $n = 2^k$ into two halves of equal cardinality and proceed recursively with each of the two halves.
- This leads to a complete binary tree M_n of depth k having the points $u_0, \ldots, u_{n-1}$ as leaves.
- Let $m_i = x - u_i$ and define each non-leaf node in the binary tree as a polynomial multiplication

$$M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \cdots m_{j \cdot 2^i + 2^i - 1} = \prod_{0 \le \ell < 2^i} m_{j \cdot 2^i + \ell}.$$

- For each level of the binary tree smaller than a threshold H, we compute the subproducts using plain multiplication.
- For each level of the binary tree larger than the threshold H, we compute the subproducts using FFT-based multiplication.

Subinverse tree

- Associated with the subproduct tree M_n ($n = 2^k$), the subinverse tree $InvM_n$ is a complete binary tree with the same height as M_n.
- For j-th node of level i in $InvM_n$ for $0 \leq i \leq k$, $0 \leq j < 2^{k-i}$,

$$InvM_{i,j} \cdot rev_{2^i}(M_{i,j}) \equiv 1 \bmod x^{2^i},$$

where $rev_{2^i}(M_{i,j}) = x^{2^i} M_{i,j}(1/x)$.

Multi-point evaluation & interpolation: estimates

Given a univariate polynomial $f \in \mathrm{K}[x]$ of degree less than $n = 2^k$ and evaluation points $u_0, \ldots, u_{n-1} \in \mathrm{K}$, compute $(f(u_0), \ldots, f(u_{n-1}))$. (Recall Recall the threshold H in the subproduct (subinverse) tree.)

- the work is $\mathrm{O}(n \log_2^2(n) + n \log_2(n) + n\, 2^H)$,
- the span is $\mathrm{O}(\log_2^2(n) + \log_2(n) + 2^H)$, and
- the parallelism overhead is $\mathrm{O}((\log_2^2(n) + \log_2(n) + H)\, U)$.

Given distinct points $u_0, \ldots, u_{n-1} \in \mathrm{K}$ and arbitrary values $v_0, \ldots, v_{n-1} \in \mathrm{K}$, compute the unique polynomial $f \in \mathrm{K}[x]$ of degree less than $n = 2^k$ that takes the value v_i at the point u_i for all i.

- the work is $\mathrm{O}(n \log_2^2(n) + n \log_2(n) + n\, 2^H)$,
- the span is $\mathrm{O}(\log_2^2(n) + \log_2(n) + 2^H)$, and
- the parallelism overhead is $\mathrm{O}(\log_2^2(n) + \log_2(n) + H)$.

Multi-point evaluation & interpolation: benchmarks

Deg.	Evaluation			Interpolation		
	CUMODP	FLINT	SpeedUp	CUMODP	FLINT	SpeedUp
2^{14}	0.2034	0.17		0.2548	0.22	
2^{15}	0.2415	0.41	1.6971	0.3073	0.53	1.7242
2^{16}	0.3126	0.99	3.1666	0.4026	1.26	3.1294
2^{17}	0.4285	2.33	5.4375	0.5677	2.94	5.1780
2^{18}	0.7106	5.43	7.6404	0.9034	6.81	7.5379
2^{19}	1.0936	12.63	11.5484	1.3931	15.85	11.3768
2^{20}	1.9412	29.2	15.0420	2.4363	36.61	15.0268
2^{21}	3.6927	67.18	18.1923	4.5965	83.98	18.2702
2^{22}	7.4855	153.07	20.4486	9.2940	191.32	20.5851
2^{23}	15.796	346.44	21.9321	19.6923	432.13	21.9441

Table: Running time (in sec.) on NVIDIA Tesla C2050 for multi-point evaluation and interpolation: CUMODP vs FLINT.

Outline

GPU support for bivariate system solving

Bivariate polynomial system solver (based on the theory of *regular chains*)

- Polynomial subresultant chains are calculated in CUDA.
- Univariate polynomial GCDs are computed in C either by means of the plain Euclidean algorithm or an asymptotically fast algorithm.

System	Pure C	Mostly CUDA code	SpeedUp
dense-70	5.22	0.50	10.26
dense-80	6.63	0.77	8.59
dense-90	8.39	1.16	7.19
dense-100	19.53	1.80	10.79
dense-110	21.41	2.57	8.33
dense-120	25.71	3.48	7.39
sparse-70	0.89	0.31	2.81
sparse-80	3.64	1.18	3.09
sparse-90	3.13	0.92	3.40
sparse-100	8.86	1.20	7.38

Table: Running time (in sec.) for bivariate system solving over a small prime field

Outline

Summary

www.cumodp.org